IWKS 3300: NAND to Tetris
Spring 2019

John K. Bennett

Compiler I: Syntax Analysis

Foundations of Global Networked Computing:
Building a Modern Computer From First Principles

This course is based upon the work of Noam Nisan and Shimon Schocken. More information can be found at (www.nand2tetris.org).
Where We Are

We are here

Human Thought

Abstract design
Chapters 9, 12

H.L. Language & Operating Sys.
Chapters 10 - 11

Compiler

Virtual Machine
Chapters 7 - 8

VM Translator

Assembly Language

Assembler

Chapter 6

Computer Architecture
Chapters 4 - 5

Hardware Platform

Chips & Logic Gates

Chips & Logic Gates

Electrical Engineering

Physics
Modern compilers are two-tiered:

- **Front-end:** from high-level language to some intermediate language
- **Back-end:** from the intermediate language to binary code.

![Diagram of modern compilers]

- **CISC machine language**
  - CISC machine
  - VM implementation over CISC platforms

- **RISC machine language**
  - RISC machine
  - VM imp. over RISC platforms

- **Intermediate code**

- **Some language**
  - Some compiler

- **Some Other language**
  - Some Other compiler

- **Jack language**
  - Jack compiler
  - VM imp. over the Hack platform

- **Any computer**
  - Hack computer
  - VM (Projects 7-8)

- **CISC machine**
  - VM imp. over CISC platforms

- **RISC machine**
  - VM imp. over RISC platforms

- **Hack machine language**
  - written in a high-level language

- **Modern compilers** are two-tiered:
  - **Front-end:** from high-level language to some intermediate language
  - **Back-end:** from the intermediate language to binary code.
Compiler Front End Architecture

- **Syntax analysis**: understanding the semantics implied by the source code
  - **Tokenizing**: creating a stream of “atoms”
  - **Parsing**: matching the atom stream to the language grammar

XML output = one way to demonstrate that the syntax analyzer works

- **Code generation**: reconstructing the program semantics using the syntax of the target code.
Tokenizing / Lexical Analysis

- Remove white space
- Construct a token list (language atoms)
- Things to worry about:
  - Language specific rules: e.g. how to treat “++”
  - Language-specific classifications: keyword, symbol, identifier, integerConstant, stringConstant, ...
- While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).
Source code

if (x < 153) {let city = "Paris";}

Tokenizer’s output

<tokens>
  <keyword> if </keyword>
  <symbol> ( </symbol>
  <identifier> x </identifier>
  <symbol> &lt; </symbol>
  <integerConstant> 153 </integerConstant>
  <symbol> ) </symbol>
  <symbol> { </symbol>
  <keyword> let </keyword>
  <identifier> city </identifier>
  <symbol> = </symbol>
  <stringConstant> Paris </stringConstant>
  <symbol> ; </symbol>
  <symbol> } </symbol>
</tokens>
The tokenizer discussed thus far is part of a larger program called *parser*.

Each language is characterized by a *grammar*. The parser is implemented to recognize this grammar in given texts.

The parsing process:

- A text stream is provided and tokenized (dealing with comments, white space, etc.)
- The parser determines the extent to which the provided text stream can be recognized in the context of the grammar
- In the process, the parser performs a complete structural analysis of the text stream

The text stream can be in an expression in a:

- Natural language (English, …)
- Programming language (Jack, …).
(5+3)*2 - sqrt(9*4)

she discussed math with her teacher
More Examples of Challenging Parsing

Time flies like an arrow

We gave the monkeys the bananas because they were hungry

We gave the monkeys the bananas because they were over-ripe

I never said she stole my money

I never said she stole my money

I never said she stole my money

I never said she stole she stole my money

I never said she stole my money

I never said she stole my money

I never said she stole my money

I never said she stole my money

John gave Pete a book.

John gave Pete a hard time.

John gave Pete a black eye.

John gave in.

John gave up.

John gave it a go.

John gave a good accounting of himself.

John’s legs gave out.

Give it up for John.

It is 300 miles, give or take a few.
Simple (terminal) forms / complex (non-terminal) forms

Grammar = a set of rules that describe how to construct complex language structures from simpler structures

Highly recursive.

A Typical Grammar of a Typical C-like Language

```
program:     statement;

statement:   whileStatement
             | ifStatement
             | // other statement possibilities ...
             | '{' statementSequence '}'

whileStatement: 'while' '(' expression ')' statement

ifStatement:  simpleIf
             | ifElse

simpleIf:    'if' '(' expression ')' statement

ifElse:      'if' '(' expression ')' statement
             'else' statement

statementSequence: '' // null, i.e. the empty sequence
                    | statement ';' statementSequence

expression:   // definition of an expression comes here

// more definitions follow
```

Code sample

```
while (expression) {
    if (expression) {
        statement;
    }
    while (expression) {
        statement;
        if (expression) {
            statement;
        }
        while (expression) {
            statement;
        }
    }
}

if (expression) {
    statement;
    if (expression) {
        statement;
    }
    while (expression) {
        statement;
    }
    if (expression) {
        statement;
    }
    if (expression) {
        statement;
    }
}
```
while (count<=100) {
/** demonstration */
count++;
// ...
}
Recursive Descent Parsing

... 
statement: whileStatement  
| ifStatement   
| ...         // other statement possibilities follow 
| '{'statementSequence '}'

whileStatement: 'while' 'expression' 'while' 'expression' statement

ifStatement: ... // if definition comes here

statementSequence: 'null, i.e. the empty sequence'  
| statement ';'; statementSequence

expression: ... // definition of an expression comes here

... // more definitions follow

- Highly recursive
- LL(0) grammars: the first token determines the applicable rule
- In most grammars, you have to look ahead 1 or more tokens
- Jack is almost LL(0).

Parser implementation: a set of parsing methods, one for each rule:
- parseStatement()
- parseWhileStatement()
- parseIfStatement()
- parseStatementSequence()
- parseExpression().
## The Jack Grammar

### Lexical elements:
The Jack language includes five types of terminal elements (tokens):

- **keyword:**
  - `class`
  - `constructor`
  - `function`
  - `method`
  - `field`
  - `static`
  - `var`
  - `int`
  - `char`
  - `boolean`
  - `void`
  - `true`
  - `false`
  - `null`
  - `this`
  - `let`
  - `do`
  - `if`
  - `else`
  - `while`
  - `return`

- **symbol:**
  - `{'`
  - `'}`
  - `('`
  - `)'
  - `['`
  - `']`
  - `.'`
  - `','`
  - `';`
  - `+'`
  - `-`
  - `'*`
  - `/`
  - `&`
  - `'|'
  - `<`
  - `'>'
  - `='`
  - `~`

- **integerConstant:** A decimal number in the range 0 to 32767.

- **StringConstant:** A sequence of Unicode characters not including double quote or newline """

- **identifier:** A sequence of letters, digits, and underscore `_` not starting with a digit.

### Program structure:
A Jack program is a collection of classes, each appearing in a separate file.

The compilation unit is a class. A class is a sequence of tokens structured according to the following context-free syntax:

- **class:**
  - `class` `className` `{'` `classVarDec*` `subroutineDec*` `'}`

- **classVarDec:**
  - `(static` `|` `field`)` `type` `varName` `,` `varName*` `';`

- **type:**
  - `int`
  - `char`
  - `boolean`
  - `className`

- **subroutineDec:**
  - `(constructor` `|` `function` `|` `method)` (`void` `|` `type`) `subroutineName`
    - `{'` `parameterList` `'}` `subroutineBody`

- **parameterList:**
  - `(type` `varName` `,` `type` `varName*)`?

- **subroutineBody:**
  - `{'` `varDec*` `statements` `'}`

- **varDec:**
  - `var` `type` `varName` `,` `varName*` `';`

- **className:** `identifier`

- **subroutineName:** `identifier`

- **varName:** `Identifier`

### Notation

- `x`: x appears verbatim
- `x`: x is a language construct
- `x?`: x appears 0 or 1 times
- `x*`: x appears 0 or more times
- `x|y`: either x or y appears
- `(x,y)`: x appears, then y.
The Jack Grammar (cont.)

<table>
<thead>
<tr>
<th>Statements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>statements: statement*</td>
</tr>
<tr>
<td>statement: letStatement</td>
</tr>
<tr>
<td>letStatement: 'let' varName ('[' expression ']'?)? '=' expression ';'</td>
</tr>
<tr>
<td>ifStatement: 'if' ('expression') '{' statements '}' ('else' '{' statements '}')?</td>
</tr>
<tr>
<td>whileStatement: 'while' ('expression') '{' statements '}'</td>
</tr>
<tr>
<td>doStatement: 'do' subroutineCall ';'</td>
</tr>
<tr>
<td>ReturnStatement: 'return' expression? ';'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression: term (op term)*</td>
</tr>
<tr>
<td>term: integerConstant</td>
</tr>
<tr>
<td>subroutineCall: subroutineName '(' expressionList ')'</td>
</tr>
<tr>
<td>expressionList: (expression (',' expression)*)?</td>
</tr>
<tr>
<td>op: '+'</td>
</tr>
<tr>
<td>unaryOp: '-'</td>
</tr>
<tr>
<td>KeywordConstant: 'true'</td>
</tr>
</tbody>
</table>

'x': x appears verbatim

x: x is a language construct

x?: x appears 0 or 1 times

x*: x appears 0 or more times

x|y: either x or y appears

(x,y): x appears, then y.
Class Bar {
    method Fraction foo(int y) {
        var int temp; // a variable
        let temp = (xxx+12)*-63;
        ...
    }
}

Syntax analyzer

- Using the language grammar, a programmer can write a syntax analyzer program (parser)
- The syntax analyzer takes a source text file and attempts to match it on the language grammar
- If successful, it can generate a parse tree in some structured format, e.g. XML.

The syntax analyzer’s algorithm shown in this slide:

- If xxx is non-terminal, output:
  <xxx>
  Recursive code for the body of xxx
  </xxx>

- If xxx is terminal (keyword, symbol, constant, or identifier), output:
  <xxx>
  xxx value
  </xxx>
**Jack Tokenizer**: a tokenizer for the Jack language (proposed implementation)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>input file/stream</td>
<td>--</td>
<td>Opens the input file/stream and gets ready to tokenize it.</td>
</tr>
<tr>
<td>hasMoreTokens</td>
<td>--</td>
<td>Boolean</td>
<td>Do we have more tokens in the input?</td>
</tr>
<tr>
<td>advance</td>
<td>--</td>
<td>--</td>
<td>Gets the next token from the input and makes it the current token. This method should only be called if <code>hasMoreTokens()</code> is true. Initially there is no current token.</td>
</tr>
<tr>
<td>tokenType</td>
<td>--</td>
<td>KEYWORD, SYMBOL, IDENTIFIER, INT_CONST, STRING_CONST</td>
<td>Returns the type of the current token.</td>
</tr>
<tr>
<td>keyword</td>
<td>--</td>
<td>CLASS, METHOD, FUNCTION, CONSTRUCTOR, INT, BOOLEAN, CHAR, VOID, VAR, STATIC, FIELD, LET, DO, IF, ELSE, WHILE, RETURN, TRUE, FALSE, NULL, THIS</td>
<td>Returns the keyword which is the current token. Should be called only when <code>tokenType()</code> is KEYWORD.</td>
</tr>
</tbody>
</table>
### Jack Tokenizer (cont.)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>Char</td>
<td>Returns the character which is the current token. Should be called only when</td>
</tr>
<tr>
<td>identifier</td>
<td>String</td>
<td>Returns the identifier which is the current token. Should be called only when</td>
</tr>
<tr>
<td>intVal</td>
<td>Int</td>
<td>Returns the integer value of the current token. Should be called only when</td>
</tr>
<tr>
<td>stringVal</td>
<td>String</td>
<td>Returns the string value of the current token, without the double quotes.</td>
</tr>
</tbody>
</table>
class SquareGame
{
    constructor SquareGame new()
    {
        if (key = 81)
        {
            let exit = true;
        }
        if (key = 90)
        {
            do square.decSize();
        }
        return this;
    }
}
Tokenizer Example

```
<class>
/// Line 1: class SquareGame {
  <keyword> class </keyword>
  <identifier> SquareGame </identifier>
  <symbol> { </symbol>
/// Line 2: constructor SquareGame new() {
  <subroutineDec>
    <keyword> constructor </keyword>
    <identifier> new </identifier>
    <symbol> ( </symbol>
    <parameterList> </parameterList>
    <symbol> ) </symbol>
  </subroutineDec>
  <subroutineBody>
    <symbol> { </symbol>
/// Line 3: if (key = 81) {
  <statements>
    <ifStatement>
      <keyword> if </keyword>
      <symbol> ( </symbol>
      <expression>
        <term>
          <identifier> key </identifier>
        </term>
        <symbol> = </symbol>
        <term>
          <integerConstant> 81 </integerConstant>
        </term>
      </expression>
      <symbol> ) </symbol>
    </ifStatement>
     <statements>
...```
The Book’s XML Tag Generation (see pp. 208-211)

**XML Tags are created for:**
- keyword
- symbol
- integerConstant
- StringConstant
- identifier
- class
- classVarDec
- subroutineDec
- parameterList
- subroutineBody
- varDec
- statements
- letStatement
- ifStatement
- whileStatement
- doStatement
- ReturnStatment
- expression
- term
- expressionList

**XML Tags are not created for:**
- type
- className
- subroutineName
- varName
- statement
- subroutineCall
- op
- unaryOp
- KeywordConstant
void WriteXML(string tag, string value) {
    if (DoXML) // global bool for enabling XML output
    {
        for (int i = 0; i < xmlIndent; i++) XMLFile.Write("    ");
        XMLFile.Write("<" + tag + "> ");
        // get rid of quotes around string constants
        // fix the XML special characters, if any
        if (value == "") value = tokenizer.currentToken;
        if (tag == "symbol")
        {
            value = value.Replace("&", "&amp;");
            value = value.Replace("<", "&lt;");
            value = value.Replace(">", "&gt;");
            value = value.Replace("\"", "&quot;");
        }
        XMLFile.Write(value);
        XMLFile.WriteLine(" </" + tag + ">");
    }
}
The **CompilationEngine** effects the actual compilation output.

- It gets its input from the Jack Tokenizer and emits its parsed structure into an output filestream.

- The output is generated by a series of `compilexxx()` routines, one for every syntactic element `xxx` of the Jack grammar.

- The contract between these routines is that each `compilexxx()` routine should read the syntactic construct `xxx` from the input, `advance()` the tokenizer exactly beyond `xxx`, and output the parsing of `xxx`. Thus, `compilexxx()` may only be called if indeed `xxx` is the next syntactic element of the input.

- In the first version of the compiler (Project 10), this module emits a structured printout of the code, wrapped in XML tags (defined in the specs of Project 10). In the final version of the compiler, this module generates executable VM code (defined in the specs of Project 11).

- In both cases, the parsing logic and module API are exactly the same.
## Compilation Engine (cont.)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Input stream/file</td>
<td>--</td>
<td>Creates a new compilation engine with the given input and output. The next routine called must be compileClass().</td>
</tr>
<tr>
<td></td>
<td>Output stream/file</td>
<td>--</td>
<td></td>
</tr>
<tr>
<td>CompileClass</td>
<td>--</td>
<td>--</td>
<td>Compiles a complete class.</td>
</tr>
<tr>
<td>CompileClassVarDec</td>
<td>--</td>
<td>--</td>
<td>Compiles a static declaration or a field declaration.</td>
</tr>
<tr>
<td>CompileSubroutine</td>
<td>--</td>
<td>--</td>
<td>Compiles a complete method, function, or constructor.</td>
</tr>
<tr>
<td>compileParameterList</td>
<td>--</td>
<td>--</td>
<td>Compiles a (possibly empty) parameter list, not including the enclosing “()”.</td>
</tr>
<tr>
<td>compileVarDec</td>
<td>--</td>
<td>--</td>
<td>Compiles a var declaration.</td>
</tr>
<tr>
<td>Method</td>
<td>Parameters</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>compileStatements</td>
<td>--</td>
<td>Compiles a sequence of statements, not including the enclosing &quot;{}&quot;.</td>
<td></td>
</tr>
<tr>
<td>compileDo</td>
<td>--</td>
<td>Compiles a do statement.</td>
<td></td>
</tr>
<tr>
<td>compileLet</td>
<td>--</td>
<td>Compiles a let statement.</td>
<td></td>
</tr>
<tr>
<td>compileWhile</td>
<td>--</td>
<td>Compiles a while statement.</td>
<td></td>
</tr>
<tr>
<td>compileReturn</td>
<td>--</td>
<td>Compiles a return statement.</td>
<td></td>
</tr>
<tr>
<td>compileIf</td>
<td>--</td>
<td>Compiles an if statement, possibly with a trailing else clause.</td>
<td></td>
</tr>
<tr>
<td>CompilationEngine (cont.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CompileExpression</strong></td>
<td>--</td>
<td>--</td>
<td>Compiles an expression.</td>
</tr>
<tr>
<td><strong>CompileTerm</strong></td>
<td>--</td>
<td>--</td>
<td>Compiles a <em>term</em>. This routine is faced with a slight difficulty when trying to decide between some of the alternative parsing rules. Specifically, if the current token is an identifier, the routine must distinguish between a variable, an array entry, and a subroutine call. A single look-ahead token, which may be one of &quot;[&quot;, &quot; (&quot;, or &quot;.&quot; suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.</td>
</tr>
<tr>
<td><strong>CompileExpressionList</strong></td>
<td>--</td>
<td>--</td>
<td>Compiles a (possibly empty) comma-separated list of expressions.</td>
</tr>
</tbody>
</table>
### The Jack Grammar (cont.)

**Statements:**

<table>
<thead>
<tr>
<th>Statement Type</th>
<th>Grammar Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>statements</td>
<td>statement*</td>
</tr>
<tr>
<td>statement</td>
<td>letStatement</td>
</tr>
<tr>
<td>letStatement</td>
<td>'let' varName ('[' expression ']')? '=' expression ';'</td>
</tr>
<tr>
<td>ifStatement</td>
<td>'if' '(' expression ')' '{' statements '}' ('else' '{' statements '}')?</td>
</tr>
<tr>
<td>whileStatement</td>
<td>'while' '(' expression ')' '{' statements '}'</td>
</tr>
<tr>
<td>doStatement</td>
<td>'do' subroutineCall ';'</td>
</tr>
<tr>
<td>returnStatement</td>
<td>'return' expression '?' ;</td>
</tr>
</tbody>
</table>

**Expressions:**

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Grammar Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>term (op term)*</td>
</tr>
<tr>
<td>term</td>
<td>integerConstant</td>
</tr>
<tr>
<td>subroutineCall</td>
<td>subroutineName '(' expressionList ')'</td>
</tr>
<tr>
<td>expressionList</td>
<td>(expression (',' expression)*)?</td>
</tr>
<tr>
<td>op</td>
<td>'+'</td>
</tr>
<tr>
<td>unaryOp</td>
<td>'-'</td>
</tr>
<tr>
<td>KeywordConstant</td>
<td>'true'</td>
</tr>
</tbody>
</table>

- **'x'**: x appears verbatim
- **x**: x is a language construct
- **x?**: x appears 0 or 1 times
- **x***: x appears 0 or more times
- **x|y**: either x or y appears
- **(x,y)**: x appears, then y.
void CompileTerm()
{
    WriteXMLTag("term");
    while (true) // make sure all code paths lead to "break"
    {
        if (GetNextToken())
        {
            if (tokenizer.tokenType == Tokenizer.TokenType.INT_CONST)
            {
                WriteXML("integerConstant", tokenizer.currentToken);
                break;
            }
            else if (tokenizer.tokenType == Tokenizer.TokenType.STRING_CONST)
            {
                WriteXML("stringConstant", tokenizer.currentToken);
                break;
            }
            else if (tokenizer.currentToken == "true")
            {
                WriteXML("keyword", tokenizer.currentToken);
                break;
            }
            else if (tokenizer.currentToken == "false")
            {
                WriteXML("keyword", tokenizer.currentToken);
                break;
            }
            else if (tokenizer.currentToken == "null")
            {
                WriteXML("keyword", tokenizer.currentToken);
                break;
            }
            else if (tokenizer.currentToken == "this")
            {
                WriteXML("keyword", tokenizer.currentToken);
                break;
            }
        }
    }
    ...
}
else if (tokenizer.tokenType == Tokenizer.TokenType.IDENTIFIER)
    {
        // save this token and peek ahead; remember that we peeked
        savedToken = tokenizer.currentToken;
        savedTokenType = tokenizer.tokenType;
        if (GetNextToken())
        {
            peekedAhead = true;
            if (tokenizer.currentToken == "[")
            {
                // array dec
                WriteXML("identifier", savedToken);
                WriteXML("symbol", tokenizer.currentToken);
                peekedAhead = false;
                CompileExpression("]");
                break;
            }
            else if (tokenizer.currentToken == ".")
            {
                // subroutine call
                WriteXML("identifier", savedToken);
                WriteXML("symbol", tokenizer.currentToken);
                peekedAhead = false;
                CompileSubroutineCall();
                break;
            }
            else
            {
                // varName; the current token points to the symbol following
                WriteXML("identifier", savedToken);
                break;
            }
        }
    }
else if (tokenizer.currentToken == "-")
{
    // unaryOp term
    WriteXML("symbol", tokenizer.currentToken);
    CompileTerm();
    break;
}
else if (tokenizer.currentToken == ".")
{
    // unaryOp term
    WriteXML("symbol", tokenizer.currentToken);
    CompileTerm();
    break;
}
else if (tokenizer.currentToken == "(")
{
    // ( expression )
    WriteXML("symbol", tokenizer.currentToken);
    CompileExpression("));
}
else ReportError("term", tokenizer.currentToken);
break;
else ReportError("term", "Premature EOF");
break;
}
WriteXMLTag("/term");
Summary and Next Steps

- **Syntax analysis:** understanding syntax
- **Code generation:** constructing semantics

The code generation (Project 11) challenge:

- Extend the syntax analyzer into a full-blown compiler that, instead of generating passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.
Perspective

- The parse tree can be constructed on the fly

- Syntax analyzers can be built using:
  - **Lex** tool for tokenizing
  - **Yacc** tool for parsing
  - Do everything from scratch (our approach ...)

- The Jack language is intentionally simple:
  - Statement prefixes: **let**, **do**, ...  
  - No operator priority
  - No error checking
  - Basic data types, etc.

- Richer languages require more powerful compilers

- **The Jack compiler**: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to follow-on courses / independent study

- Industrial-strength compilers:
  - Have excellent error diagnostics
  - Generate efficient code
  - (Usually) support parallel (multi-core) processors.